UMAPIZZA

Aaron Cettolin, Giorgio Colomban, Luca Carminati
Supervisor: Marco Domenico Santambrogio
University: Politecnico di Milano
Team number: xohw18-267

Introduction

The current tech landscape is heavily characterized by the huge amount of data that has to analyzed by automatic algorithm. Machine Learning is one of the most fastly growing field in Computer Science, and has encountered a serious obstacle in the so called “Curse of Dimensionalty”, that is the fact that complex events are described through an exponential number of variable, but such huge amounts of information aren't manipulable yet by current algorithms on current architectures.
While creating faster algorithms is important, and it has been done for years by the academical community, optimizing the existing algorithms for specific architectures is crucial to obtain the most efficient solutions in order to save time and money.

That’s why we decided to implement one of the latest dimensionality reduction algorithms, UMAP (Uniform Manifold Approximation and Projection), exploiting the possibilities and advantages of the FPGA-based architectures, to make the core portion the algorithm run as fast as possible on server-based services, where the incremented efficiency can bring the biggest advantages, due to the amount of data that gets processed in those kinds of applications.

The final goal of the project is to make our accelerated version more versatile and useful for everyone involved in the field, from software engineers, with a transparent interface to call the optimized functions through a python framework, to computer engineers, that will work on accelerating UMAP in the next year, giving them our work as a baseline. En fact, at the moment this document is written, the only available UMAP implementation is in Python and no other implementations are publicly available.

All our designs are Open Source, available at the following link

Design

In this chapter, a description of the PIZZA team project's implementation is provided.

We started by profiling the original Python algorithm to understand which function should be optimized, with a big canonical dataset as an input, the full MNIST, to get relevant results.

Function	Runtime (seconds)	Runtime (% of total runtime)	Number of calls
nn_descent	375.054	58%	1
euclidean_random_projection_split	134.305	21%	148429
optimize_layout	95.472	15%	1
other functions	32.801	6%	22019352

The profiling results showed that a single call to the function nn_descent took more than 58% of the total runtime, making it the perfect candidate for optimization. Its Role in the UMAP ecosystem(?) is that of find in an approximate way the K Nearest Neighbors of each data point in order to support the Local Manifold Approximation at the basis of the algorithm.
The nn_descent function first got rewritten in C along with auxiliary functions that are used inside nn_descent, divided in 3 files:

```
* utils.c: various auxiliary functions
* heap_utilis.c: functions to create, modify and delete heaps
* nn_descent.c: the core function
```

Each function has been optimized to be easily implementable on hardware architectures.

Most of the variable have been declared with custom types, to allow easy reconfiguration in each situation where a different type is needed

C Implementation

This subchapter will include a description of each function in each file, along with the relative code snippets

utils.c

- **tau_rand_int**:

  The function takes an array as an input uses to compute a pseudo-random integer, using bitwise operations, as shown in the original UMAP code.

```
int tau_rand_int(int state[N_STATES])   {

    // rng_state is an array that represents the internal state of the rng
    state[0] = (((state[0] & 4294967294) << 12) & 0xffffffff) ^ ((((state[0]
<< 13) & 0xffffffff) ^ state[0]) >> 19);
    state[1] = (((state[1] & 4294967288) << 4) & 0xffffffff) ^ ((((state[1]
<< 2) & 0xffffffff) ^ state[1]) >> 25);
    state[2] = (((state[2] & 4294967280) << 17) & 0xffffffff) ^ ((((state[2]
<< 3) & 0xffffffff) ^ state[2]) >> 11);

    return state[0]^state[1]^state[2];
}
```

- **tau_rand**

  The function pseudo-randomly returns 1 or 0 by calling *tau_rand_int* and checking its sign.

```
int tau_rand(int state[]) {
// Similar to the previous function, but it returns a int between 0 and 1
int rng;
rng = tau_rand_int(state);

if(rng > 0){
  return 1; //50% of chance of selecting it
}
return 0;
}
```

- **rejection_samples**

  The function populates the results array by randomly picking *n_samples* samples from a *pool_size* wide pool. The random selection is performed by calling *tau_rand.*

```
void rejection_sample(int n_samples,int pool_size, int rng_state[], t_index
result[]) {
    // n_samples: number of random samples to pick from the pool of samples
    // pool_size: size of the pool of samples
    // rng_state: state of the random vector (similar to tau_rand's arg)

    int i, k;
    t_index j;
    boolean reject_sample = 0;
```

```
    for(i = 0; i < n_samples; i++)  {
        reject_sample = 1;
        while(reject_sample == 1)   {
            j = _abs_t_index(tau_rand_t_index(rng_state)) % pool_size;
            for(k = 0; k < i; k++)  {
                if(j == result[k])
                    break;
            }
            if(j != result[k])
                reject_sample = 0;
        result[i] = j;
        }
    }

}
```

- **build_candidates**

  The function takes every point's neighbors' heap and generates a list of the possible data points candidate to update each point's neighbor list.

```
void build_candidates(
    t_index current_graph_index[N_POINTS*K_NEIGHBORS],
    boolean current_graph_flag[N_POINTS*K_NEIGHBORS],
    int rng_state[],
    t_index candidate_neighbors_index[N_POINTS*MAX_CANDIDATES],
    t_distance candidate_neighbors_dist[N_POINTS*MAX_CANDIDATES],
    boolean candidate_neighbors_flag[N_POINTS*MAX_CANDIDATES]
    ){
    int i, j, idx, isn;
    int d;

    for(i = 0; i < N_POINTS; i++) {
        for(j = 0; j < K_NEIGHBORS; j++) {
            // if(current_graph_index[i*K_NEIGHBORS + j] < 0)   // The
function should work without this if statement
            //  continue;
            idx = current_graph_index[i*K_NEIGHBORS + j];
            isn = current_graph_flag[i*K_NEIGHBORS + j];
            d = tau_rand(rng_state); //Remove tau_rand when changing
build_candidates

heap_push_candidate(candidate_neighbors_index,candidate_neighbors_dist,candi
date_neighbors_flag, i, d, idx, isn);

heap_push_candidate(candidate_neighbors_index,candidate_neighbors_dist,candi
date_neighbors_flag, idx, d, i, isn);
            current_graph_flag[i*K_NEIGHBORS + j] = 0;
        }
    }
}
```

- **euclidean_distance**

The function simply computes the euclidean distance between two N dimensional points.

```
t_distance euclidean_distance(t_data x[N_FEATURES],t_data y[N_FEATURES]){
    t_distance result=0;

    for(t_index i=0; i<N_FEATURES; i++){
            result += (((t_distance)x[i]-(t_distance)y[i]))*
((t_distance)x[i]-(t_distance)y[i]);
    }
    return result;
}
```

heap_utils.c

- **heap_push**

  The function takes a heap of neighbors of a point and a new possible neighbor and pushes it into the heap.

```
void heap_push_opt(
    t_index current_graph_index[N_POINTS*K_NEIGHBORS],
    t_distance current_graph_dist[N_POINTS*K_NEIGHBORS],
    boolean current_graph_flag[N_POINTS*K_NEIGHBORS],
    int row,                                        //determines which data
point we are addressing
    t_distance weight,                                   //determines the
distance (for heap sorting)
    t_index index,                                     //is the element to
add
    boolean flag,                                      //determines whether
this is to be considered a new addition
    int *c
)
{
    t_distance cur_elem_dist = weight;
    t_index cur_elem_index = index;
    boolean cur_elem_flag = flag;

    int current_element; //local max
    int current_son; //left son
    int current_daughter; //right son
    int idx1;
    int idx2;
    boolean found=0; //boolean

    //do not insert first element if we already have this element
    for (idx1 = 0; idx1 < K_NEIGHBORS; idx1++)
    {
        if (index == current_graph_index[row*K_NEIGHBORS + idx1])
        {
            found= 1;
        }
    }

    //try to insert val in position 0
```

```
    if(!found && current_graph_dist[row*K_NEIGHBORS + 0] > weight){
        swap_t_index(&current_graph_index[row*K_NEIGHBORS + 0],
&cur_elem_index);
        swap_t_distance(&current_graph_dist[row*K_NEIGHBORS + 0],
&cur_elem_dist);
        swap_boolean(&current_graph_flag[row*K_NEIGHBORS + 0],
&cur_elem_flag);
        *c = *c + 1;
    }

    //recreate the heap push (to have more regularity this is done the same
number of times (ln(K_NEIGHBORS)) to have more regularity)
    current_element = 0;
    current_son = 1;
    current_daughter = 2;
    //while (current_son < K_NEIGHBORS) //not unrollable
    for(int i=0; i< L2_K_NEIGHBORS-1; i++) //unrollable //arrive till the
level before the last (so current element is always < K_NEIGHBOR)
    {
        if(current_daughter<K_NEIGHBORS
            && current_graph_dist[row*K_NEIGHBORS + current_daughter] >
current_graph_dist[row*K_NEIGHBORS + current_son]
            && current_graph_dist[row*K_NEIGHBORS + current_daughter] >
current_graph_dist[row*K_NEIGHBORS + current_element]){

            idx1 = current_daughter;
            idx2 = current_element;
            current_element = current_daughter;
            current_son = current_daughter*2+1;
            current_daughter = current_son+1;
        }
        else if(current_son<K_NEIGHBORS
            &&current_graph_dist[row*K_NEIGHBORS + current_son] >
current_graph_dist[row*K_NEIGHBORS + current_element]){

            idx1 = current_son;
            idx2 = current_element;
            current_element = current_son;
            current_son = current_son*2+1;
            current_daughter = current_son +1;
        }
        else //fake case in which no swap is to be made (current element is
a leaf or no child is more distant than current element)
        {
            idx1 = current_element;
            idx2 = current_element;
            current_element = current_son;
            current_son = current_son*2+1;
            current_daughter = current_son+1;
        }
        swap_t_index(&current_graph_index[row*K_NEIGHBORS + idx1],
&current_graph_index[row*K_NEIGHBORS + idx2]);
        swap_t_distance(&current_graph_dist[row*K_NEIGHBORS + idx1],
&current_graph_dist[row*K_NEIGHBORS + idx2]);
        swap_boolean(&current_graph_flag[row*K_NEIGHBORS + idx1],
&current_graph_flag[row*K_NEIGHBORS + idx2]);
    }
}
```

- **heap_push_candidate**

  Very similar to the previous function, but it works on *N_POINTS*MAX_CANDIDATES* long arrays, rather than *N_POINTS*K_NEIGHBOURS* long ones. (See *defines.h* for more details)

```
int heap_push_candidate(
    t_index current_graph_index[N_POINTS*MAX_CANDIDATES],
    t_distance current_graph_dist[N_POINTS*MAX_CANDIDATES],
    boolean current_graph_flag[N_POINTS*MAX_CANDIDATES],
    int row,                                        //determines which data
point we are addressing
    t_distance weight,                                 //determines the
distance (for heap sorting)
  t_index index,                                     //is the element to add
    boolean flag                                     //determines whether
this is to be considered a new addition
)
{
    //break if the new element is more distant than the actual max
    if (weight >= current_graph_dist[row*MAX_CANDIDATES +0])
    {
        return 0;
    }

    int idx;
    //break if we already have this element
    for (idx = 0; idx < MAX_CANDIDATES; idx++)
    {
        if (index == current_graph_index[row*MAX_CANDIDATES +idx])
        {
            return 0;
        }
    }

    // insert val at position zero
    current_graph_dist[row*MAX_CANDIDATES +0] = weight;
    current_graph_index[row*MAX_CANDIDATES +0] = index;
    current_graph_flag[row*MAX_CANDIDATES +0] = flag;

    // descend the current_graph, swapping values until the max heap
criterion is met
    int i = 0;
    int i_swap; //temp index to swap with the i-th
    int ic1; //position 1
    int ic2; //position 2

    while (1)
    {
        ic1 = 2 * i + 1;
        ic2 = ic1 + 1;

        //determine whether the i-th element has to be swapped with one of
his son
        if (ic1 >= MAX_CANDIDATES)
            break;
        else if (ic2 >= MAX_CANDIDATES)
```

```
        {    //if the distance in position ic1 is more than the distance
passed to the function
            if (current_graph_dist[row*MAX_CANDIDATES +ic1] > weight)
                i_swap = ic1; //temp equals to the position ic1
            else
                break;
        }
        //if the distance in position ic1 is >= of the distance in position
ic2
        else if (current_graph_dist[row*MAX_CANDIDATES +ic1] >=
current_graph_dist[row*MAX_CANDIDATES +ic2])
        {     //if the distance in position ic1 is more than the distance
passed to the function
            if (weight < current_graph_dist[row*MAX_CANDIDATES +ic1])
            {
                i_swap = ic1; //temp equals to the position ic1
            }
            else
            {
                break;
            }
        }
        else
        {     //if the distance in position ic2 is more than the distance
passed to the function
            if (weight < current_graph_dist[row*MAX_CANDIDATES +ic2])
                i_swap = ic2; //temp equals to the position ic2
            else
                break;
        }

        //do the swap
        current_graph_dist[row*MAX_CANDIDATES +i] =
current_graph_dist[row*MAX_CANDIDATES +i_swap];
        current_graph_index[row*MAX_CANDIDATES +i] =
current_graph_index[row*MAX_CANDIDATES +i_swap];
        current_graph_flag[row*MAX_CANDIDATES +i] =
current_graph_flag[row*MAX_CANDIDATES +i_swap];

        i = i_swap;
    }
     //"i" has the last i_swap's value of the cycle 'while'
     //in that 'i' position the programm insert the values passed to the
function
    current_graph_dist[row*MAX_CANDIDATES +i] = weight;
    current_graph_index[row*MAX_CANDIDATES +i] = index;
    current_graph_flag[row*MAX_CANDIDATES +i] = flag;

    return 1; //The number of new elements successfully pushed into the
heap.
}
```

- **deheap_sort**

The function takes an max-heap as input and puts all its elements in a array, sorted by the elements' weights. It's really similar to the second half of a classic heap sort, the first part it's not necessary because the input is already a max-heap.

```
  void deheap_sort(
  t_index current_graph_index[N_POINTS * K_NEIGHBORS],
  t_distance current_graph_dist[N_POINTS * K_NEIGHBORS],
  boolean current_graph_flag[N_POINTS * K_NEIGHBORS],
  t_index result_index[N_POINTS*K_NEIGHBORS],
  t_distance result_dist[N_POINTS*K_NEIGHBORS]
  ){

  for(int n_point=0; n_point< N_POINTS; n_point++){

      for(int i_neighbor=0; i_neighbor < K_NEIGHBORS; i_neighbor++){
 //TODO: per i_neighbor = K_NEIGHBORS - 1 (ultima iterazione), questo fa
schifo
          //swap the maximum with the last element and rebuild the heap
          //so the next iteration the current maximum will be ignored
(already at its place)
          swap_t_index(&current_graph_index[n_point*K_NEIGHBORS + 0],
&current_graph_index[n_point*K_NEIGHBORS + K_NEIGHBORS-1-i_neighbor]);
          swap_t_distance(&current_graph_dist[n_point*K_NEIGHBORS + 0],
&current_graph_dist[n_point*K_NEIGHBORS + K_NEIGHBORS-1-i_neighbor]);
          swap_boolean(&current_graph_flag[n_point*K_NEIGHBORS + 0],
&current_graph_flag[n_point*K_NEIGHBORS + K_NEIGHBORS-1-i_neighbor]);

          //Restore the heap property for a heap with an out of place
element in position 0

 siftdown(current_graph_index,current_graph_dist,current_graph_flag,
K_NEIGHBORS-1-i_neighbor , n_point, 0);

          //put the values of the current_heap inside the result in the same
position
          result_dist[n_point*K_NEIGHBORS + K_NEIGHBORS-1-i_neighbor] =
current_graph_dist[n_point*K_NEIGHBORS + K_NEIGHBORS-1-i_neighbor];
          result_index[n_point*K_NEIGHBORS + K_NEIGHBORS-1-i_neighbor] =
current_graph_index[n_point*K_NEIGHBORS + K_NEIGHBORS-1-i_neighbor];
      }
  }
}
```

- **siftdown**

  The function takes a max heap as input and rearranges it if an element is out of place, to restore the max heap properties.

```
void siftdown(t_index current_graph_index[N_POINTS * K_NEIGHBORS],
  t_distance current_graph_dist[N_POINTS * K_NEIGHBORS],
  boolean current_graph_flag[N_POINTS * K_NEIGHBORS],
  int size_heap,
  int row,
  int elt){

  int left_child;
  int right_child;
```

```
  int swap; //temp for the swapping

  while(elt*2+1 < size_heap){
      left_child = elt *2+1;
      right_child= left_child+1;
      swap = elt; //temp equals to the position of the out of place element
      /*if the dist value of the current_heap in position 'swap' is less
than the value in position 'left_child'
       ==> swap equals to the 'left_child'*/
      if(current_graph_dist[row*K_NEIGHBORS + swap] <
current_graph_dist[row*K_NEIGHBORS + left_child]){
          swap = left_child;
      }
      /*if right_child is less than size_heap &&
      if the dist value of the current_heap in position 'swap' is less than
the value in position 'right_child'
       ==> swap equals to the 'right_child'*/
      if (right_child < size_heap && current_graph_dist[row*K_NEIGHBORS +
swap] < current_graph_dist[row*K_NEIGHBORS + right_child]){
          swap = right_child;
      }
      //if the swap value is equal to the 'elt' value passed to the function
==> break
      if(swap == elt){
          break;
      }
      else{
          //call a function by address for the swapping
          swap_t_index(&current_graph_index[row*K_NEIGHBORS + elt],
&current_graph_index[row*K_NEIGHBORS + swap]);
          swap_t_distance(&current_graph_dist[row*K_NEIGHBORS + elt],
&current_graph_dist[row*K_NEIGHBORS + swap]);
          swap_boolean(&current_graph_flag[row*K_NEIGHBORS + elt],
&current_graph_flag[row*K_NEIGHBORS + swap]);

          elt = swap;
      }
  }
}
```

headers.h

The header file *headers.h* contains the prototypes for the main functions, all the defines for the code and all the custom type definitions, that this way can be kept in the same place and changed just once to affect the whole program.

defines.h

The header file *defines.h* contains all the definitions for the program. The main ones are the following:

- **MAX_DIST**: the maximum square distance value that can be calculated
- **MAX_CANDIDATES**: the maximum number of candidates considered when searching for a point's neighbors
- **RHO**: the dropout probability of a heap_push (in nn_descent algorithm a push marked to be executed can be randomly not be executed)

- **N_ITERS**: the number of iterations of the main core of nn_descent
- **DELTA**: a coefficient used to determine dynamically when to stop the execution of nn_descent with respect to the number of successful pushes executed
- **N_STATES**: the size of the rng_state array. At the moment this value is fixed to 3
- **N_POINTS**: number of points considered
- **N_FEATURES**: number of dimensions for each of the N_POINTS
- **K_NEIGHBOURS**: number of neighbors calculated for each point
- **L2_K_NEIGHBOURS**: the logarithm base 2 of the K_NEIGHBORS (calculated automatically)
- **N_LEAFS**: number of leaves in the rp_tree (computed automatically)
- **LEAF_SIZE**: number of features for each leaf (computed automatically)

nn_descent.c

This is the main function of the program. It takes an array of data as an input and uses the previously described functions to find neighbors for each point, using a heap structure to store the intermediate results. The basic algorithm is to get, for each point, his neighbors' new neighbors and try to push them in his heap. This operation is repeated until the result converges to a local optimum and too few updates are performed in each cycle.

A little optimization done is to drop (ie not execute) a push scheduled to be executed with a probability of RHO. This optimization statistically does not make the results significantly less reliable, as shown in the original NN Descent paper [W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures"].

##

```
void nn_descent(
        t_data data_in[N_POINTS*N_FEATURES],
        //K_NEIGHBORS
        int rng_state_in[N_STATES],
        //MAX_CANDIDATES
        //N_ITERS
        //DELTA
        //RHO
        //rp_tree_init considered True
#if RP_TREE_INIT
        t_index leaf_array_in[N_LEAF * LEAF_SIZE],
#endif
        t_index result_out_index[N_POINTS*K_NEIGHBORS],
        t_distance result_out_dist[N_POINTS*K_NEIGHBORS]
){
DO_PRAGMA(HLS INTERFACE m_axi depth=N_POINTS*N_FEATURES port=data_in
offset=slave bundle=hostmem)
DO_PRAGMA(HLS INTERFACE m_axi depth=N_STATES port=rng_state_in offset=slave
bundle=hostmem)
DO_PRAGMA(HLS INTERFACE m_axi depth=N_POINTS*K_NEIGHBORS port=result_out_index
offset=slave bundle=hostmem)
DO_PRAGMA(HLS INTERFACE m_axi depth=N_POINTS*K_NEIGHBORS port=result_out_dist
offset=slave bundle=hostmem)
#if RP_TREE_INIT
DO_PRAGMA(HLS INTERFACE m_axi depth=N_LEAF*LEAF_SIZE port=leaf_array_in
offset=slave bundle=hostmem)
#pragma HLS INTERFACE s_axilite port=leaf_array_in bundle=control
#endif
```

```
#pragma HLS INTERFACE s_axilite port=data_in bundle=control
#pragma HLS INTERFACE s_axilite port=rng_state_in bundle=control
#pragma HLS INTERFACE s_axilite port=result_out_index bundle=control
#pragma HLS INTERFACE s_axilite port=result_out_dist bundle=control

#pragma HLS INTERFACE s_axilite port=return bundle=control

    //utility variables/array used to statically save intermediate results
    t_distance d;
    static t_index candidate_neighbors_index[N_POINTS*MAX_CANDIDATES];
    static t_distance candidate_neighbors_dist[N_POINTS*MAX_CANDIDATES];
    boolean candidate_neighbors_flag[N_POINTS*MAX_CANDIDATES];

    int c;
    t_index p;
    t_index q;
    static t_index current_graph_index[N_POINTS * K_NEIGHBORS];
    static t_distance current_graph_dist[N_POINTS * K_NEIGHBORS];
    boolean current_graph_flag[N_POINTS * K_NEIGHBORS];

    //burst memory access to data
    static t_data data[N_POINTS * N_FEATURES];
    static t_index result_index[N_POINTS*K_NEIGHBORS];
    static t_distance result_dist[N_POINTS*K_NEIGHBORS];
    int rng_state[N_STATES];
    memcpy(data,(const t_data*)data_in, N_POINTS*N_FEATURES*sizeof(t_data));
    memcpy(rng_state, (const int*)rng_state_in, N_STATES*sizeof(int));
#if RP_TREE_INIT
    static t_index leaf_array[N_LEAF*LEAF_SIZE];
    memcpy(leaf_array, (const t_index*)leaf_array_in,
N_LEAF*LEAF_SIZE*sizeof(t_index));
#endif
    //in substitution of make_heap we initialize directly our heap
    for(int n_points=0; n_points < N_POINTS; n_points++){
        for(int i_neighbor = 0; i_neighbor< K_NEIGHBORS; i_neighbor++){
                current_graph_index[n_points * K_NEIGHBORS + i_neighbor] = -1;
                current_graph_dist[n_points * K_NEIGHBORS + i_neighbor] =
MAX_DIST;
                current_graph_flag[n_points * K_NEIGHBORS + i_neighbor] = 0;
        }
    }

    //push of random points in each point's heap (~random initialization)
    static t_index indices[K_NEIGHBORS];

    if(VERBOSE) printf("Random Inizialization Start\n");

    //cycle on each data point
    for(int i=0; i< N_POINTS; i++){
        //create an array of random indices used to select randomly some element
from data to initialize the heaps
        rejection_sample(K_NEIGHBORS, N_POINTS, rng_state, indices); //what
happens if i is in indices?

        for(int j=0; j<K_NEIGHBORS; j++){
            //compute the distance for the current pair of point and push them
in each other's heap
```

```
            d = euclidean_distance( &data[i*N_FEATURES],
&data[indices[j]*N_FEATURES]);
            heap_push(current_graph_index, current_graph_dist,
current_graph_flag, i, d, indices[j], 1);
            heap_push(current_graph_index, current_graph_dist,
current_graph_flag, indices[j], d, i, 1);
        }
    }

    if(VERBOSE) printf("Random Inizialization End\n");
    if(VERBOSE) printf("RPTree Inizialization Start\n");

    //push in each point's heap the approximate neighbor derived from RPTree (~
heuristic initialization)
    // How do RPTree work? https://www.youtube.com/watch?v=UFE2XtAaCck Minute
1:48
#if RP_TREE_INIT
    //cycle on each leaf array
    for(int m=0; m<N_LEAF; m++){

        //cycle on each element of each leaf
        for(int i=0; i< LEAF_SIZE; i++){
            //are elements not initialized of the leaf initialized to -1?
            //they are not considered
            if (leaf_array[m*LEAF_SIZE + i] < 0){
                break;
            }
            //cycle on all successor elements in the leaf
            for(int j=i+1; j<LEAF_SIZE; j++){
                if(leaf_array[m*LEAF_SIZE + j] < 0){
                    break;
                }
                //compute distances between each pair (i,j) of points
                d= euclidean_distance(&data[leaf_array[m*LEAF_SIZE + i] *
N_FEATURES], &data[leaf_array[m*LEAF_SIZE +j] * N_FEATURES]);

                heap_push(current_graph_index, current_graph_dist,
current_graph_flag, leaf_array[m*LEAF_SIZE + i], d, leaf_array[m*LEAF_SIZE + j],
1);
                heap_push(current_graph_index, current_graph_dist,
current_graph_flag, leaf_array[m*LEAF_SIZE + j], d, leaf_array[m*LEAF_SIZE + i],
1);
            }
        }
    }

#endif

    if(VERBOSE) printf("RPTree Inizialization End\n");
    if(VERBOSE) printf("Main Block Start\n");

    //main block of NNDescent
    //in each iteration we scan the neighbors' neighbors and (if not already
compared before) we try to push them in the current point heap
    //OSS: we consider each point singularly and then confront all its neighbors
between them tryin to push each one in the other
    //(a point neighbors are neighbor's neighbor each one of the other)
```

```
    //for each iteration of the algorithm
    for(int n=0; n<N_ITERS; n++){

        if(VERBOSE) printf("Iteration n° %d\n", n);
        //initialize the array of potential candidate neighbor for each array

        for(int n_point=0; n_point < N_POINTS; n_point++){
            for(int i_neighbor = 0; i_neighbor< MAX_CANDIDATES; i_neighbor++){
                    candidate_neighbors_index[n_point*MAX_CANDIDATES +
i_neighbor] = -1;
                    candidate_neighbors_dist[n_point*MAX_CANDIDATES +
i_neighbor] = MAX_DIST;
                    candidate_neighbors_flag[n_point*MAX_CANDIDATES +
i_neighbor] = 0;
            }
        }

        build_candidates(current_graph_index, current_graph_flag,
                         rng_state, candidate_neighbors_index,
candidate_neighbors_dist, candidate_neighbors_flag);

        if(VERBOSE) printf("Candidates Built\n");

        c=0;

        //cycle on each data point ~ i is the index of the point we are
analyzing
        for(int i=0; i<N_POINTS; i++){
//#pragma HLS PIPELINE II=1

            //cycle on each candidate neighbor of the i-th point
            for(int j=0; j<MAX_CANDIDATES; j++){

                //p is the index of the j-th candidate neighbor data point
                p= candidate_neighbors_index[i*MAX_CANDIDATES + j];

                //are elements not initialized of the leaf initialized to -1?
                //ignore element with negative index
                //randomly ignore the 50% of candidate neighbor ~ Dropout to get
better performances
                if(p<0 || tau_rand(rng_state) < RHO){
                    continue;
                }

                //cycle on each candidate neighbor of the i-th point
                for(int k=0; k<MAX_CANDIDATES;k++){

                    //q is the index of the k-th candidate neighbor data point
                    q = candidate_neighbors_index[i*MAX_CANDIDATES + k];

                    //ignore element with negative index
                    //ignore the pair in which the two are both flagged not new
('0')
                    if(q<0 || (!candidate_neighbors_flag[i*MAX_CANDIDATES + j]
&& !candidate_neighbors_flag[i*MAX_CANDIDATES + k])){
                        continue;
                    }
```

```
                    d=
euclidean_distance(&data[p*N_FEATURES],&data[q*N_FEATURES]);

                    //c represents the number of updates executed during the
iteration
                    // try to push p, q in each other's graph
                    //c+= heap_push(current_graph_index, current_graph_dist,
current_graph_flag, p, d, q, 1);
                    //c+= heap_push(current_graph_index, current_graph_dist,
current_graph_flag, q, d, p, 1);
                    heap_push_opt(current_graph_index, current_graph_dist,
current_graph_flag, q, d, p, 1, &c);
                    heap_push_opt(current_graph_index, current_graph_dist,
current_graph_flag, q, d, p, 1, &c);
                }
            }

        }
        if(VERBOSE) printf("N° of updates: %d\n\n",c);
        //if too few updates happened, the algorithm stops
        if(c<= DELTA * K_NEIGHBORS * N_POINTS){
            break;
        }
    }
    //remove the heap structure ~linearize the array of our final result
    deheap_sort(current_graph_index, current_graph_dist, current_graph_flag,
result_index, result_dist);

    //burst memory access to result_out
    memcpy(result_out_index,(const t_index*)result_index,
N_POINTS*K_NEIGHBORS*sizeof(t_index));
    memcpy(result_out_dist,(const t_distance*)result_dist,
N_POINTS*K_NEIGHBORS*sizeof(t_distance));
}
```

FPGA Implementation

During the course of this project, we developed different workflows based on different tools used. This sub-chapter will include a description of the main parts of each workflow used.

We developed 2 Workflows:

- Vivado HLS Workflow
- SDAccel Workflow

In both the Workflows a python script, *main.py,* has been used to automatically generate the correct inputs and defines passed to our program. *main.py* is based on a slightly modified version of nn_descent, in order to be able to extract variables and outputs for our tests.

main.py automatically generates the following files, previously described:

- *defines.h*
- *headers.h*
- *signals.h*

In order to execute *main.py*, one has to define some internal defines, that affect the behavior of *nn_descent,* accordingly to the type of computation desired.

This script has the function of having an automatic translation from python to C of the computation.

This and the following steps are needed to generate a bitstream that is runnable on the FPGA of choice, against data of exact dimensions specified.

One of the next steps of the project will be that a bitstream able to perform the operation on any data if the dimension is less than the max allowed.

Step by step instructions for each workflow are given to the user in order to permit the intended use of the various parts of the project and to give a smooth experience of adaptation of our code to everyone's needs.

After the generation of the custom C sources, the user can follow the usual **Vivado HLS → Vivado → SDK** workflow. The C sources generated are already synthesizable in an exportable IP Core, that will have to be imported into Vivado where the Core can be integrated with the FPGA architecture. The final bitstream can be exported in SDK and we provide the file *helloworld.c* useful to perform its testing.

If the user prefers to perform higher level synthesis, he can use the SDAccel workflow. Some little modifications (included in the README) have to be performed on the code, then the OpenCL kernel is synthesizable. The host file is provided as well, *test-cl.cpp.* A custom makefile can be used to produce the final executable.

Testing

3 different Scenario-of-Use Tests were executed on 2 different boards, a xc7z020clg400-1 ZYNQ board and a xilinx:adm-pcie-ku3:2ddr-xpr:4.0-0 Kintex UltraScale board, and their timings have been compared to a comparable CPU hardware. In the case of the ZYNQ, the CPU benchmark was obtained on @PC1, whereas the Kintex was compared to @PC2

@PC1 Specs: ~ Hp Pavillion Laptop

```
*OS: Arch Linux x86_64*

*Host: HP Pavilion Laptop 14-bf1xx*

*Kernel: 5.1.15-arch1-1-ARCH*

*CPU: Intel i7-8550U (8) @ 4.000GHz*

*Memory: 7890MiB*
```

@PC2 Specs: ~ PowerEdge Server

```
*OS: CentOS Linux release 7.6.1810 (Core)*

*Host: PowerEdge R720xd*

*Kernel: 3.10.0-957.12.2.el7.x86_64*

*CPU: Intel Xeon E5-2680 v2 (40) @ 3.600G*

*Memory: 386754MiB*
```

Each hardware execution test was performed 20 times in order to attenuate the intrinsic hardware timing variance. The median timing is shown as result.

A brief description of the Use Case Scenarios of the tests:

TEST 1

ZYNQ vs Normal Laptop

In this test we compared a ZYNQ against a normal laptop, in order to simulate a low-budget scenario, where the computation can only be executed locally

TEST 2

KintexUltraScale vs Server - Memory Test

In this test we compared a Kintex Ultrascale against a powerful Server machine. The scenario is that of a high-budget centralized computation center. The particular issue addressed by this test is that of high memory usage, maximizing out the memory usage.

TEST 3

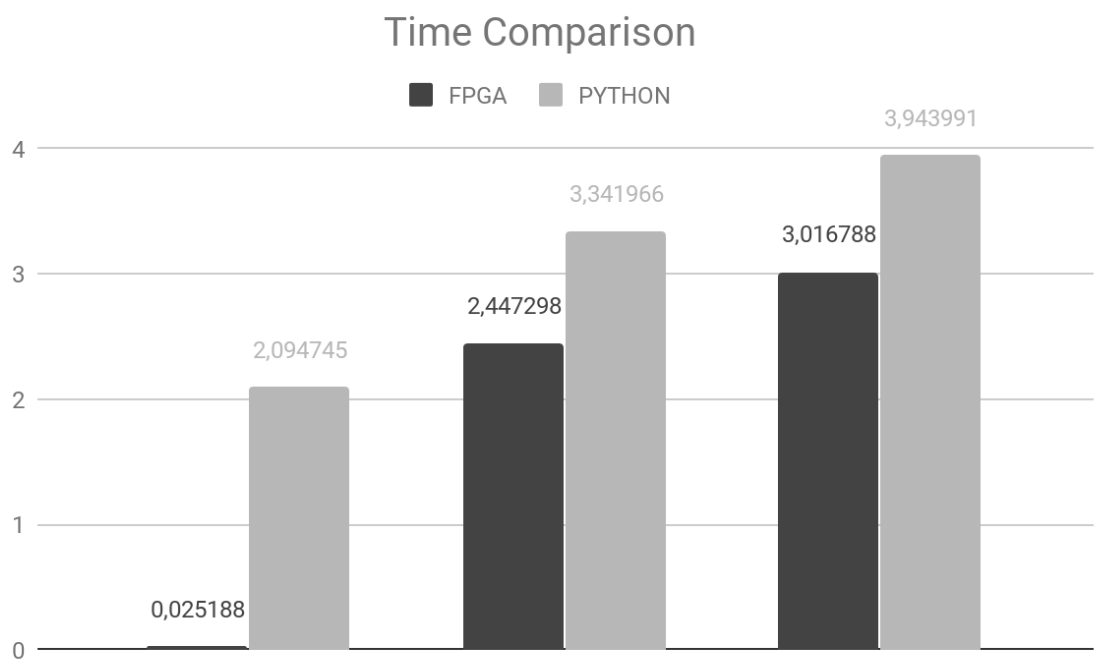
KintexUltraScale vs Server - Complexity Test

In this test we compared a Kintex Ultrascale against a powerful Server machine. The scenario is that of a high-budget centralized computation center. The particular issue addressed by this test is that of high computation load, due to the high number of K Neighbors requested

Results and Graphical Comparison

Name	TEST 1	TEST 2	TEST 3
N_POINTS	100	4000	2600
N_FEATURES	6	12	5
MIN_COORD	0	0	0
MAX_COORD	1000	10000	10000
K_NEIGHBORS	15	14	50
MAX_CANDIDATES	30	30	30
N_LEAF	56	3683	703
LEAF_SIZE	15	14	50
Time	Median	Median	Median
FPGA	0,025188	2,447298	3,016788
Input transfer	0,000003	0,00076	0,000654
Computation	0,025184	2,107486	2,676463
Output Transfer	0,000001	0,001207	0,001798
PYTHON	2,094745	3,341966	3,943991
Speedup	83,16440368	1,365573788	1,307347749

*: in Computation time of Test 1 the time of FPGA programming isn't included, whereas in Test 2 and 3 it is included



Evaluation of the results

- Our FPGA implementation is faster than the original python one and with less variance in the results
- Memory Transfers do not use a large amount of time
- Memory occupation is the bottleneck of our implementation. The limited amount of BRAMs available on FPGA limits strongly the possibility of testing bigger test cases where the complexity is much higher and the configuration time overhead is ignorable. As seen in the differences between Test 1 and 2/3 the configuration is a strong overhead in respect to the real computation time. Even using custom data types with the least possible bits, the memory occupied is too large to scale up the amount of data.

In “Conclusions & Future Work” chapter we will define the next steps to address this issue.

Conclusions and future work

As said in the introduction, we plan to expand the existing python framework, used at the moment for hardware prototyping and testing. We think that a well structured python framework could be very useful for software developers, allowing them to have a transparent way to call the optimized function in larger application, by just importing a library that will wrap the lower level code.

However our current workflow can be already useful for those experts that want to optimize UMAP more than done in our implementation.

At the lower level the project can be extended with great achievements regarding memory usage, for example using an external memory that streams the data to a computational core that has only little saved data even if this will mean having a different implementation than the original python one.

Our work has demonstrated that important speedups are achievable in this context, even if being in an early stages of development and not useful yet in real-case scenarios.